Energy Efficient Architecture for Graph Analytics Accelerators

ISCA’16

Mustafa Ozdal*, Serif Yesil*, Taemin Kim†, Andrey Ayupov†, John Greth†, Steven M. Burns†, Ozcan Ozturk*

* Bilkent University, Ankara, Turkey
† Intel Corporation, Oregon, USA
Dark silicon era

Accelerator rich architectures: Customized hardware for specific applications

Hardware design is complex and time consuming

Many applications. Which ones to accelerate? Months of design effort.

Template based design: Capture commonalities for a domain
Graph Analytics

- Model relationships between individual entities

- Emerging application areas:
  Social networks, web, recommender systems, ...

- Example applications: PageRank, Collaborative Filtering, Loopy Belief Propagation, Betweenness Centrality, ...

- Graph-level parallelism & iterative algorithms
Graph Accelerator Template

Targeted Graph Computation Pattern:
- Vertex-centric & Gather - Apply - Scatter (GAS)

We propose:
- Energy efficient accelerator architecture for irregular graph applications
- Well-defined template to plug in different applications
- Synthesizable SystemC models for architecture exploration & hardware generation

Design Productivity & Efficiency:
- Template code size: 39K lines, user code size 43 lines for PageRank
- PageRank: 65X better power efficiency than 24 cores of Xeon CPU
Outline

- Targeted Application Characteristics
- Graph-Parallel Abstraction
- Proposed Architecture
- Experimental Results
Graph Analytics

Different than traditional HPC
- Irregular data access & communication
- Poor cache locality
- Computation-to-communication ratio very low
- Irregular topologies due to scale-free graphs

Convergent algorithms
- Throughput vs. work-efficiency
- Different implementation choices
- High throughput easier to achieve than work efficiency
Asymmetric Convergence

Processing all vertices in every iteration is not work-efficient!

About 2x more edges processed for PageRank!

Synchronous vs. Asynchronous Execution

Jacobi iteration formula for PageRank:

\[ r^{k+1}(v) = \left( \frac{1 - \alpha}{N} \right) + \alpha \sum_{(u \rightarrow v)} \frac{r^k(u)}{\text{degree}(u)} \]

Synchronous: All vertices are updated simultaneously.

Gauss-Seidel iteration formula for PageRank:

\[ r^{k+1}(v) = (1 - \alpha) + \alpha \sum_{u < v} \frac{r^{k+1}(u)}{\text{degree}(u)} + \alpha \sum_{u > v} \frac{r^k(u)}{\text{degree}(u)} \]

Asynchronous: Updates to a vertex are visible to others in the same iteration. **Observed to be much faster to converge! (30-50% less work)**
Throughput vs. Work Efficiency

Asymmetric Convergence

- Process all vertices
  - Easier to implement
  - **High throughput**
  - Worse work efficiency

- Process active vertices only
  - Maintain worklist, dynamic work assignment
  - Lower throughput
  - **Better work efficiency**

Iterative Execution Model

- **Synchronous**
  - Easier to implement
  - **High throughput**
  - Worse work efficiency

- **Asynchronous**
  - Fine-grain synchronization, sequential consistency support
  - Lower throughput
  - **Better work efficiency**

Ozdal, et. al. ICCAD 2015
Outline

- Targeted Application Characteristics
- Graph-Parallel Abstraction
- Proposed Architecture
- Experimental Results
Gather-Apply-Scatter Abstraction

- Abstraction proposed by Graphlab for distributed computing (Low, et. al. VLDB 2012)
- Data structures associated with each vertex and edge
- Compute operations defined for 3 stages of a vertex program:

1. **GATHER**
   - Data structures associated with each vertex and edge

2. **APPLY**
   - Compute operations defined for 3 stages of a vertex program:

3. **SCATTER**
Outline

- Targeted Application Characteristics
- Graph-Parallel Abstraction
- Proposed Architecture
- Experimental Results
**ACCELERATOR UNIT**

- **Active List Mgr**: Maintains active vertices
- **Runtime**: Schedules vertex computation
- **Gather Unit**: Accumulates data from neighbors for a vertex
- **Apply Unit**: Performs main computation for a vertex using gather results
- **Scatter Unit**: Distributes the new data to neighbors; activates neighbors
- **Memory modules**: Customized per graph data type
Compute Units

Gather Unit
- Neighbor vertices and edges accessed. **Poor cache locality!**
- Latency tolerant: Tens of vertices and hundreds of edges processed concurrently. **High MLP!**
- Storage for partial vertex and edge states with dynamic load balancing
- Dependency between neighboring vertices handled through Sync Unit

Apply Unit
- Computation done on local data only

Scatter Unit
- Similar to Gather Unit
- Memory writes in addition to reads
- Neighbor vertex activations
Control Units

**Sync Unit**
- Ensures race-free and sequentially-consistent execution of vertices
- Maintains execution states of vertices and assigns a *rank* for each vertex
- Guarantees the proper RAW and WAR ordering for neighboring vertices
- High-throughput processing

**Active List Manager**
- Active vertices stored in main memory with efficient caching
- High-throughput access mechanisms
- Race-free simultaneous accessed without explicit locks
- Coordinates with Sync Unit for asynchronous execution
Multiple Accelerator Units

- Banked design: Each unit responsible for a static subset of vertices
- Two global light-weight modules:
  - **GTD**: Global Termination Detector
  - **GRC**: Global Rank Counter
Outline

- Targeted Application Characteristics
- Graph-Parallel Abstraction
- Proposed Architecture
- Experimental Results
Benchmarks

Applications
- PageRank (PR)
- Single Source Shortest Path (SSSP)
- Stochastic Gradient Descent (SGD)
- Loopy Belief Propagation (LBP)

Datasets
- **PR & SSSP**: 6 datasets from Snap and generated with Graph500 *(up to 1B edges)*
- **LBP**: 3 images generated with GraphLab’s synthetic image generator *(up to 18M edges)*
- **SGD**: 2 movie datasets from MovieLens *(up to 10M edges)*
Experimental Setup

Baseline CPU
- 2-socket 24-core IvyBridge Xeon with 30MB LLC and 132GB of main memory
- Optimized software implementations in OpenMP/C++
- Running Average Power Limit (RAPL) to estimate energy
- Projected DDR3 power (measured) to DDR4 power (in-house DDR4 model)

Proposed Accelerator
- **Performance**: Cycle accurate SystemC model + DRAMSim2
- **Accelerator power and area**: HLS + physical-aware logic synthesis with a 22nm industrial library
- **Cache power and area**: CACTI models
- **DRAM power**: in-house DDR4 model
Performance Comparison

Accelerator Speed Up

- 24-cores
- 12-cores

Data for various tasks showing speed up when using 24-cores versus 12-cores.
Power Comparison

Accelerator power is dominated by DRAM power. Improvements would be ~10x higher without DRAM power.
Conclusions

A template architecture for graph-analytics is proposed
- Latency tolerance for irregular accesses
- Graph-parallel execution with sequential consistency
- Asynchronous execution and active vertex set support

Synthesizable and cycle-accurate SystemC models
- Different accelerators generated by plugging in app-specific functions
- Template code size: 39K lines, user code size 43 lines for PageRank

Experiments with 22nm industrial libraries:
- Performance comparable with a 24-core Xeon system (except SSSP)
- Up to 65x less power